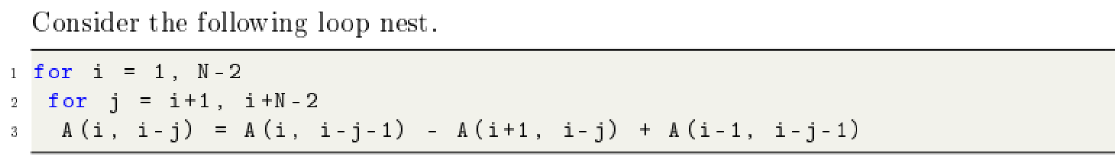
CS610 Assignment 2
Name: Manish (190477)

SOLUTION 1.



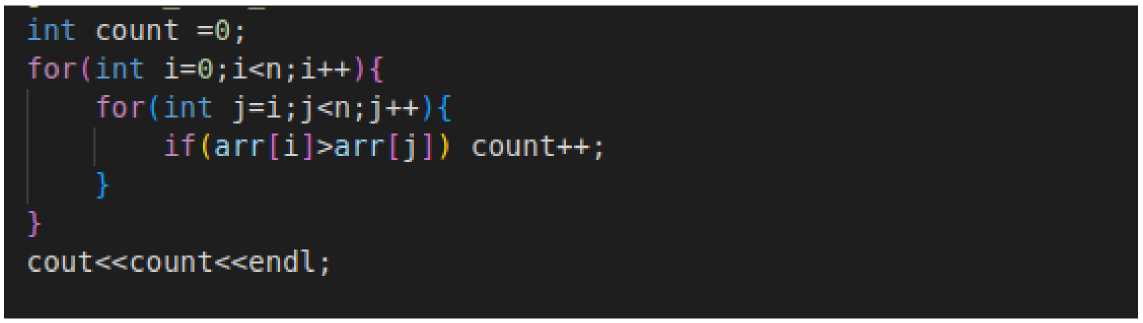
Consider the following loop nest.

```
for i = 1, N-2
 for j = i+1, i+N-2
  A(i, i-j) = A(i, i-j-1) - A(i+1, i-j) + A(i-1, i-j-1)
```

Dependency testing using Delta test:
1. Between A(i,i-j) and A(i,i-j-1):
$i = i + \Delta i \longrightarrow \Delta i = 0$
$i - j = i + \Delta i - j - \Delta j - 1 \longrightarrow \Delta j = -1$
This is Anti dependence(0,-1).
2. Between A(i,i-j) and A(i+1,i-j):
$i = i + \Delta i + 1 \longrightarrow \Delta i = -1$
$i - j = i + \Delta i - j - \Delta j \longrightarrow \Delta j = 1$
This is Anti dependence(-1,1).
3. Between A(i,i-j) and A(i-1,i-j-1)
$i = i + \Delta i - 1 \longrightarrow \Delta i = 1$
$i - j = i + \Delta i - j - \Delta j - 1 \longrightarrow \Delta j = 0$
This is flow dependence(1,0).

SOLUTION 2.

Implementation Overview:
We have given n numbers and p threads. Array id assigned with a random number between [0,n-1] using the template code given.
So normally(single core) we can count the number of inversions as follows:



```
int count =0;
for(int i=0;i<n;i++){
    for(int j=i;j<n;j++){
        if(arr[i]>arr[j]) count++;
    }
}
cout<<count<<endl;
```

What we can do to parallelize the inversion count is to divide the outer loop n iterations into p threads such that those threads get almost the same work to perform.

We can divide iterations of the outer loop as follows:
First, n%p threads get n/p+1, and the rest of the threads get n/p iterations of the outer loop.
Every thread counts the number of inversions in iterations assigned to it and returns the value.
In the parent process, we add all the returned values from each of the threads and print them.
One extra thing I am doing is when n¡p, I am only creating n threads only because p-n threads get 0 iterations to perform, So there is no point in creating more than n threads which result in performance overheads.

Code Overview:
Struct thread_args is used to pass proper arguments to a thread like from which iteration to start, array pointer, and the number of iterations to perform.
Function generate_rand_nums is used to initialize the array. (Same as the template given)
Function count_inversion is run by every thread with the proper arguments using the struct thread_args.

SOLUTION 3.

Implementation Overview:
We have given P producer threads, the shared buffer which can hold B values at a time, and W worker threads with the conditions stated in the questions.
So to achieve this we are going to need some variables and pthread mutex to achieve that.
1. We have to keep track of the number of elements, and start index of valid buffer elements. These variables are changed by the worker and producer thread to check conditions and update them accordingly. So to achieve this mutex1 is used so that these variables are accessed and updated correctly by threads
2. We have to read from the file line by line. So to achieve this correctly we have to use mutex2.
3. We have to keep track of the number of producer threads exited because if all the threads are exited means all the lines are read and written to buffer correctly. After reading the complete buffer, the work of the worker thread should be finished as no producer thread is present to write to the buffer. So to achieve this I used the terminate_signal variable. It should be updated carefully. So mutex3 is used to do it correctly.
4. Writing to file should be done properly so that it should not write on the previous values. So to achieve that mutex4 is used when the number is prime and we are writing that number to the file. The rest of the things are done in parallel like checking prime, initialization, etc.
Code Overview:
Function prime: Return true if a number is prime otherwise false.
Function write_to_ buffer: This function is executed by producer threads. In this function, we initialize variable parallel. Using mutex2 we read a line until the whole file is read. Once the line is read and converted to int. Now we check if the number_of_elements in the buffer is less than B(using mutex1), If yes then we write it to the buffer and update the variables accordingly and read the next line otherwise we wait until there is space to write to the buffer. When the complete file is read we terminate the thread and update terminal_signal using mutex3.
Function Reading_from_buffer: This function is executed by worker threads. In this function, we initialize variable parallel. There is a while loop that does the task until there is no element in the buffer and all producer threads execution is completed. In this while loop, we take an element

from the buffer and update the variable accordingly using mutex1. We check if this number is prime or not. If this number is prime then using mutex4 we write this number to the file prime.txt.

SOLUTION 4.

Implementation Overview: We have a manager which put jobs in job_queue and 4 worker threads to do the task in it.
So to achieve this we are going to need three mutex and jd variable to keep account of job completed.
1. We have to apply one mutex to job_queue so that worker threads don't read the same job.
2. We need one mutex for hash table api to work correctly.
3. We need one mutex for jd variable which we are updating when we complete a job.

Code Overview:
Function Work_to_do: We first check if job completed is equal to total job(mutex3). If yes then we exit the loop, so from the thread. If job completed are less than total job, then we check the size of job queue. If job queue is zero then we wait for manager to write to the queue otherwise we pop a element out of the queue(mutex1). After poping the task, we decode it and do whatever is the type of the task(mutex2). After completing the task, we increase the jd variable by 1(mutex3).
Function enqueue: We check the size of the queue before en-queuing. If it is 8 we return otherwise we push the task into the queue. When we call enqueue first time ,we also have to create 4 worker thread.
Result:
Here task are performed parallel by 4 worker threads. We cannot predict the result as it depend on the scheduling of the threads.